2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

Continuous, Low Overhead, Run-Time Validation of Program Executions

Erdem Aktas*, Furat Afram**, Kanad Ghose
Computer Science Department
State University of New York at Binghamton
Binghamton, NY, USA
[eaktas, fafram1, ghose]@cs.binghamton.edu

Abstract— **The construction of trustworthy systems demands that the execution of every piece of code is validated as genuine, that is, the executed codes do exactly what they are supposed to do. Pre-execution validations of code integrity fail to detect run time compromises like code injection, return and jump-oriented programming, and illegal dynamic linking of program modules. We propose and evaluate a generalized mechanism called REV (for *R*un-time *E*xecution *V*alidator) that can be easily integrated into a contemporary out-of-order processor to validate, as the program executes, the control flow path and instructions executed along the control flow path. To prevent memory from being tainted by compromised code, REV also prevents updates to the memory from a basic block until its execution has been authenticated. Although control flow signature based authentication of an execution has been suggested before for software testing and for restricted cases of embedded systems, their extensions to out-of-order cores is a non-incremental effort from a microarchitectural standpoint. Unlike REV, the existing solutions do not scale with binary sizes, require binaries to be altered or require new ISA support and also fail to contain errors and, in general, impose a heavy performance penalty. We show, using a detailed cycle-accurate microarchitectural simulator for an out-of-order pipeline implementing the X86 ISA that the performance overhead of REV is limited to 1.87% on the average across the SPEC 2006 benchmarks.**

Keywords-component; Trusted Computing, Control-Flow Validation, Secure Execution, Control-Flow Integrity, Hardware Security, Computer Security

I. INTRODUCTION

A central requirement for implementing trusted computing platforms is to validate whether a program executing on a potentially untrusted host is really the program the user thinks it is. In an untrusted environment/host, a program may be compromised in many ways: through static or dynamic replacement of part or all of the binaries, or through static or dynamic linking to untrusted library functions, or through attacks that affect calls and returns. With any of these compromises, the program does not perform its intended functions correctly. The detection of such compromises requires the execution of the entire program to be validated at run time, including the validation of called functions in libraries, the kernel and utilities.

Validating program executions at run-time is a difficult process. Despite the use of non-executable pages [47] and address layout randomization (ASLR) [7], applications remain vulnerable and an attacker can alter the control flow to take control of the victim applications as they execute [64, 46, 22, 28, 13]. Hund et al. show that run-time attacks can bypass existing mechanisms for assuring the integrity of kernel code and install a rootkit successfully on a Windows based system [28]. Even without any code modification, adversaries can resort to code reuse attacks (CRA) to alter the control flow path by writing into structures that are used for generating a target address. For instance, attackers can call unintended functions by overwriting VTables, which hold function pointers. Similarly, return-to-lib-C attacks [52] and jump and/or return oriented programming techniques can alter control flow without overwriting the code area [9, 15, 14]. Once a kernel flaw is introduced, it is possible to disable the existing security mechanisms, gain full control of the victim system and inject code to the kernel memory space [13]. Recent APT (Advanced Persistent Threat) attacks prove that the attacker can still bypass the existing protection mechanisms by using zero-day vulnerabilities [58, 21] and/or by using their expertise on known vulnerabilities. Digital Rights Management (DRM) applications are also subject to run-time attacks that disable calls to the license verification system [22].

One way to validate the execution of a program is to continuously monitor all attempted changes/updates to the various software components in the program and the rest of the system including the operating system and libraries. The only changes that are permitted are the ones that have been certified as legitimate. Unfortunately, this approach results in a fairly closed system and program execution can be slowed down dramatically if changes to software components that are made at run-time need to be certified as the program executes.

Detecting attacks that compromise code and control flow at run-time is challenging in general. There are a plethora of solutions that detect specific types of run-time attacks against code integrity and control flow integrity (Sec. II).

---

* Currently with McAfee (Intel Security), **Currently with Intel. This work was done while the first two authors were at SUNY-Binghamton

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.18

We present and evaluate a simple hardware mechanism called REV (Run-time Execution Validator), for an out-of-order (OOO) processor to continuously validate the execution of a program - *as it executes* - to detect any compromises to the control flow and any compromise of the binaries. REV takes a more direct and *universal* approach to not only detect such attacks at run-time but *also prevents information from compromised executions from tainting the memory* and from affecting the computation state. REV is thus agnostic to specific types of attacks and as such, it is capable of detecting known attacks, zero-day attacks and unknown future attacks against control flow and code integrity with a small trusted code base (Section VII). REV detects control flow and code integrity attacks at run-time by generating an execution signature of the basic blocks (BBs) and the control flow path and by comparing such signatures against statically generated reference signatures. The reference signatures are *held in RAM in an encrypted form* to detected unwarranted changes made to them at run-time. Table 1 lists how REV detects some well-known classes of attacks.

The contributions of this paper are as follows:

- We propose the use of a signature cache (SC) and techniques to hide the delay of generating run-time execution signatures for the continuous and complete validation of a program (and the library/kernel functions that it invokes) with a low performance overhead. ISA extensions or binary modifications are not needed, nor do we require source-code access or compiler support. Also, as REV does not rely on binary modification, code reentrancy is guaranteed as well. REV validates execution at basic block boundaries and defers changes to the memory made by a basic block until its execution has been validated.
- We propose the use of secret keys to encrypt the reference signature information in memory to prevent attackers from defeating the validation mechanism.
- We show how the proposed mechanism can be used to perform different types of validations and how the reference data set used for validation can be reduced dramatically in size when limited forms of validations are needed.
- We propose the use of a new, low-overhead technique to validate the control flow path originating at a return instruction that terminates a function call. This technique does not rely on the use of a shadow call stack.

REV has been validated using a full-system cycle-accurate simulator that boots up the Linux kernel and simulates the execution of all software components on a realistic (simulated) out-of-order CPU core supporting the X86-64 ISA and all aspects of the memory hierarchy including DRAM, caches, and interconnections. Our evaluation demonstrates that the performance penalty of the scheme is negligible in most cases and tolerable in the extreme instances within the SPEC 2006 benchmark suite. We also provide some quantitative estimates of the additional hardware needed to support our techniques to show that the additional hardware investment is practically viable.

TABLE I. EXAMPLE ATTACKS AND THEIR DETECTION BY REV

Attack Type	How system is attacked	How REV Detects Attack
Direct Code Injection	Binaries are overwritten on the fly by another process, usually with higher privilege	Basic block crypto hash will not match reference hash value
Indirect code injection	Binaries overwritten because of an inherent vulnerability or new code added to the call stack is executed because of buffer overflows	Basic block crypto hash will not match reference hash value; control flow path will not match the path known from static analysis
Return Oriented Attack	Function calls return to unintended basic blocks	Control flow path will not match path known from static analysis
Jump Oriented Attack	Gadgets (pieces of code) are used to construct a desired attack code	The crypto hash of the gadget code will not match the crypto hash of the original basic blocks. Control flow path will not match the path known from static analysis.
Vtable compromises	Overwriting Vtable at runtime in object oriented programming environments to alter the control flow	Control flow path will not match path known from static analysis.
Return to lib-C attacks	Overwriting the function return address to a lib-C function address	Control flow path will not match path known from static analysis.

To the casual reader, our work may appear incremental relative to the work of [20, 25, 49, 6, 41] simply because both solutions use basic block signatures and control flow path information. REV uses microarchitectural techniques and a systems-level approach to dynamically validate both code integrity and the control flow integrity as the program executes in both user and kernel levels with only a 1.87% performance overhead on the average across the SPEC benchmarks. Since the bulk of the REV mechanism is within the CPU, REV requires a very small trusted code base at the system level.

II. RELATED WORK

The most common technique in use today for certifying an execution as genuine relies on the use of the trusted platform module [60] or a similar facility, including a stand-alone cryptoprocessor within a tamper-proof package [31], to authenticate binaries prior to their execution. These techniques fail to detect compromises that occur at run-time. The Cerium co-processor [16] validates the signature of cache lines dynamically on cache misses using a context switch to a micro-kernel, thus introducing a serious performance bottleneck. Cerium assumes that all of the trusted code will fit into the CPU cache, an impractical assumption for contemporary systems. The SP-architecture [38] can detect run-time code modifications for instructions but it does not verify the control flow and is thus incapable of detecting control flow attacks, such as return oriented programming. The SP-architecture also requires compiler support and new instructions. The AEGIS co-processor [56] removes the vulnerability of CPU-external authentication logic and authenticates the trusted parts of the kernel and

protects the local memory contents when required using one-time pads but does not facilitate efficient, continuous authentication of executing programs.

Control flow authentication (CFA) has generally relied on validating control flow paths and/or validating the contents of basic blocks of instructions using cryptographic hashes for the basic blocks. Although all of these schemes use basic block hashes [20, 25, 49, 6, 41], they differ significantly in terms of their capabilities and implementation. Dynamic code certification has been proposed for software testing. For instance, in [53], compile time signatures are inserted in the binary and are verified by a coprocessor that examines blocks of fetched instructions; the work of [41] uses basic block signatures, embedded into the literal fields of RISC instructions, for the same purpose. In both schemes, the reference signatures are stored unencrypted, so the system can be easily compromised. In [25], the authors propose the use of hash signatures on data and code blocks to authenticate the blocks as they are fetched, using an FPGA-based coprocessor that holds the reference signatures. This scheme requires compiler support and new instructions for validation of related functions.

Arora et al. [6] proposed a dynamic CFA mechanism for embedded systems. REV differs in fundamental ways from this design. The differences have to do with the storage and use of the reference signatures, scalability and applicability in general to an OOO pipeline. In [6] content-addressed memory (CAM) tables, pre-loaded by the linker/loader, are used to hold the reference signatures for basic blocks. This introduces two limitations. First, one cannot have large tables to hold all of the basic block signatures of binaries encountered in general-purpose systems. Second, context switches require table updates that are very slow and practically useless in most contemporary embedded systems. We solve these problems by using signature caches that are dynamically managed. Such caches require appropriate changes to an OOO pipeline as addressed in our proposed solution. The work of [6] also requires source code access to insert explicit control flow instructions to mark the end of a basic block. The work of [6] also does not prevent stores from an illicit basic block from modifying memory. Finally, the performance overhead of [6] makes their adaptation to a general-purpose OOO processor unattractive.

The REM [20] and IMPRES [49] mechanisms introduce techniques that use keyed hashes to authenticate basic blocks (BBs) at run time. Both techniques do not validate how control flowed into a basic block, require instruction binaries to be re-written and both require BB hashes to be regenerated when the hash key is changed. REM also requires an ISA extension. In [KZK 10], the authors propose the use of unique signatures for code traces. To avoid a performance penalty, the traces that are less likely to execute are neither explicitly identified, nor validated.

Argus [41] primarily targets reliability issues and embeds basic block signatures into unused fields of instructions in binaries of a RISC ISA. To adopt Argus for authenticating executions dynamically, the signatures will have to be encrypted and the delay of decrypting the reference signatures will have to be somehow absorbed.

Several software techniques have been proposed to verify control flow at periodic intervals or at specific checkpoint locations [53, 11, 12, 34, 55, 33, 3]. These techniques fail to detect compromises in-between two consecutive checks, relying on the assumption that any compromise can still be detected using pertinent checks at the next point of validation. Furthermore, these techniques require either compiler support or permanent modification of the executables, both of which are not required in REV. Most of these techniques also have adverse performance implications. Implementing a software solution for validating remote execution remains an open area of research and only a few practical solutions have been proposed to date. A plethora of software techniques, including the use of hardware support [68], have been explored in obfuscating control flow in a program to thwart code modifications. Control Flow Integrity (CFI) [1, 2] and its variants are software techniques for validating the run-time control flow against a statically derived control flow path. CFI relies on the generation of unique basic block ids and their insertion and run-time validation using a proprietary tool. CFI assumes the kernel and kernel components are trusted and legitimate; thus CFI is neither intended, nor able, to detect attacks at the kernel level. Rootkits can reset the NX bit and thus defeat CFI. Furthermore, CFI does not validate the instructions along the control flow path as REV does.

The technique of [18] uses the frequency of return instructions to detect the return oriented programming (ROP) [52] attacks. The work of [36] later expanded the same technique to detect return oriented programming (ROP) and jump oriented programming (JOP) [9] attacks in hardware by looking at the frequency of the computed branches and return instructions in a limited branch history window. A relatively simpler approach has been shown in [44] where the last branch prior to a system call is examined to detect a ROP attack. Unfortunately, these techniques are limited to detecting ROP/JOP attacks and do not verify code integrity or control flow integrity against other types of attacks. In [67], the authors proposed a CFI technique that uses a hybrid linear-recursive disassembly approach to disassemble binaries then use a static analysis to find possible CFI targets with a performance overhead of up to 45%.

A software technique for preventing return-oriented attacks was introduced in [19] based on the assumption that call and return instructions occur in pairs and that the scope of jumps within a function are limited to the boundary of the function and a function has one entry point. An identical technique, implemented in hardware called Branch Regulation, was introduced later in [35]. Those two techniques use a shadow stack to verify the function returns. [19] relies on a large trusted code base in kernel and require binary modification; branch regulation relies on user-supplied annotations and (presumably) static code analysis.

A low overhead technique, called control flow locking (CFL) for detecting control flow attacks is presented in [8]. CFL effectively pairs up an indirect call with a legitimate return by inserting code preceding such a call to set a lock. A legitimate return for this call also has code inserted prior to it to reset the lock. CFL fails to detect the first attack and

also fails to verify that the correct return is paired up with the call. It also requires binary modification.

Pappas et al. introduced in-place code randomization to prevent the re-use of unintended code chunks [48]. This technique replaces the code inside the basic blocks with randomly chosen but equivalent instructions, thus the unintended code chunks will be different for each compilation. For the same purpose, Wartell et al. introduced a mechanism that randomizes the basic block addresses at runtime through de-compilation and re-compilation of the binary [62]. Xia et al. introduces CFIMon in [65] to monitor the execution through the ptrace interface using the branch trace store mechanism. The work of [4] has investigated the extent to which software techniques can validate the control flow path with an acceptable performance overhead, relying on existing hardware support for branch tracing in some Intel CPUs. Orthrus [27] is a hardware-based mechanism for detecting control flow compromises that uses duplicated code chunks running on two cores to detect control flow violations and requires OS support for memory updates and mapping. Hypersafe [61] uses a non-bypassable memory lockdown mechanism and restricts pointer indexing to protect the hypervisor by reducing the attack surface exposed to ROP/JOP attacks and code injections.

As a related effort, OASIS exemplifies a recent effort to realize a safe haven for execution with the ability to support attestability [43]. OASIS uses a small trusted code base but requires the addition of new instructions and the source code to have calls to special functions built around these instructions. Intel's SGX mechanism [32] also adds new instructions to protect private code and data, again with the goal of ensuring a safe haven for execution. REV does not require any new instruction, nor the recompilation of the binaries.

III. Run-Time CFA for OOO Processors

Although architectural support for run-time control flow authentication (CFA) has been proposed and evaluated in the past [20, 41, 25, 6, 49], the proposals have mainly targeted simpler embedded systems, where the overhead for such authentication is not crippling. This is not the case with out-of-order (OOO) processors and reducing the performance overhead of run-time CFA is critical in these CPUs. Furthermore, techniques devised for embedded systems, such as the ones in [6, 25], cannot be directly extended to OOO CPUs. Specifically, one has to address the following requirements of a run-time CFA for an OOO CPU and for the proposed solution in general:

R0. The technique should be general enough to detect *any* kind of compromises to code and control flow integrity and not be limited to detecting specific instances of such attacks.

R1. The technique should be *scalable*: arbitrarily large binaries and statically or dynamically linked libraries have to be handled. Thus, one cannot store reference signatures used for validation in CPU-internal tables, as in [6].

R2. The technique should have as low performance overhead as possible. In particular, the performance overhead introduced by the generation of the cryptographic hash of a BB and for fetching, decrypting and comparing the reference signature should all be minimized.

R3. The technique should ideally be transparent to the executables. Most existing techniques require ISA extensions [20, 41, 25, 6, 49] and some require the executables to be modified.

R4. The technique should support context switching and system calls as transparently as possible without imposing a large overhead. The techniques of [6, 25] do not meet this requirement: in [6], the processor internal table holding the reference signatures has to be reloaded on every context switch.

R5. Changes to the permanent state of the system (committed registers, memory locations) made by instructions should be disallowed until the execution of these instructions are validated. Existing hardware-based CFA techniques fail to meet this important requirement.

R6. Speculative instruction execution should be supported and executions/authentications along a mispredicted path should be abortable. Existing CFA hardware fail to do this.

We now describe our technique for supporting run-time CFA in an OOO datapath and show how all of these requirements are met. Our technique can be easily retrofitted to an existing OOO design, does not require any ISA extension and guarantees full binary compatibility of all executables, including the OS and libraries. Thus, our technique meets Requirement R3.

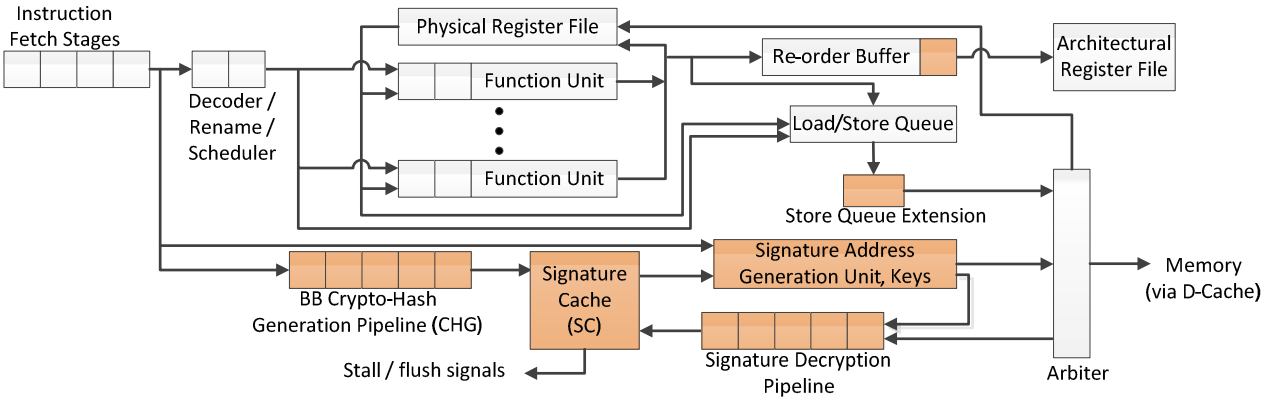

Figure 1. Out-of-Order Datapath, the components needed by REV are shaded in orange

IV. REV: RUN-TIME EXECUTIONS VALIDATOR

A. Overview

Our proposed technique for control flow authentication and the authentication of instructions executed along the control flow path is called REV: ***R***un-time ***E***xecution ***V***alidator. In REV, we validate the signature of each basic block (BB) of instructions as the control flow instruction terminating the BB (a branch, jump, return, exit etc.) is committed. In addition to comparing the cryptographic hash function of the instructions in the BB against a reference signature, we also validate that control flowed into the BB along an expected path. We do this for the computed branch addresses and calls by comparing the actual address of the target of branch/call instruction at the end of the current BB against potential target addresses stored as part of the reference signature of the BB. Thus, requirement R0 is met.

Figure 1 depicts a typical OOO datapath and it also shows the additional components needed for implementing REV (shaded). In REV, the reference signatures of the BBs and the source/destination address pairs are stored in the main memory, encrypted with a secret key. As instructions are fetched by the front-end stages, they are fed into a pipelined crypto hash generator (**CHG**), which operates in parallel with the rest of the pipeline. Because of speculative execution and potential flushing of instructions along the mispredicted path, it makes sense to only validate control flow along the path of committed instructions. This post-commit validation also permits the delay of the crypto hash generation to be effectively overlapped with the steps of instruction decoding, renaming, dispatch, issue, execution, writeback and commitment, thus meeting the first part of requirement R2. The post-commit validation also meets Requirement R6 naturally.

We incorporate a small on-chip cache, called the ***signature cache*** (**SC**), to hold reference signatures that are retrieved from memory. Because of the temporal locality of reference, the SC eliminates the need to fetch the same reference signatures repeatedly from the memory, thus improving the overall performance. As instructions are fetched by the front end, the predicted target address of a branch is used to retrieve the reference signature of the target BB from the main memory via normal memory interface (going through the L1 D-cache and the rest of the on-chip memory hierarchy).

The use of the SC enables the scalability requirement (Requirement R1) to be met: the SC will only hold the recently-used (and likely-to-be-used) reference signatures and signatures of any executing modules are loaded into the SC as the code executes, unaffected by the size or the number of code modules used. The SC is probed using the address of the instruction that terminates a BB - henceforth, we refer to this address as the "***address of the BB***".

On a SC miss, the SC fetches the reference signature for a BB from memory, going through the L2 cache and other levels of the on-chip memory hierarchy, based on the BB address. The reference signature, which is stored encrypted in the RAM, is decrypted using a secret key as it is retrieved into the SC and prior to its use in authenticating the execution of a BB. In practice, the effective SC miss handling time is reduced because of hits in the on-chip cache hierarchy and because of the delay in-between the fetching the first instruction of a BB to the time of validating the signature of the BB. Thus the second part of Requirement R2 is met.

The SC permits context switches to be handled naturally, as it automatically fetches signatures for the executing code, thus meeting Requirement R4. The CHG generates the signature of the instruction stream forming a basic block as the instructions are fetched along the path predicted by the branch predictor for the pipeline. When the last instruction of the current basic block is ready to commit, the SC is probed to verify the calculated crypto hash value of the basic block against the reference control flow signature, as stored in the SC entry for the BB. At this time, on a SC hit, followed by a match of the control flow signature for the basic block, execution continues as usual. On a SC miss, the corresponding reference signature is retrieved into the SC and till a match is performed and the pipeline is stalled. On a signature mismatch an exception is raised and appropriate handlers are invoked.

To meet Requirement R5, we extend the ROB beyond the normal commit stage and also extend the store queue similarly, as shown in Figure 1 to prevent changes to the precise state unless a BB is validated. Since these ROB extensions have a finite capacity, *we break up the very rare BBs that contain a long sequence of instructions artificially into multiple BBs*, limiting the number of stores or the total number of instructions within a BB (whichever occurs earlier). The front-end of the pipeline is aware of this limit and triggers SC lookups at these artificial boundaries instead of waiting for a control flow instruction to be fetched.

A much stricter approach to meeting Requirement R5 is to defer all changes to the system state till the entire execution has been authenticated. One way to do this is to employ the concept of page shadowing [42]. Initially, the original pages accessed by the program are mapped to a set of shadow pages with identical initial content. All memory updates are made on the shadow pages during execution and when the entire execution is authenticated, the shadow pages are mapped in as the program's original pages. Also, while execution is going on, no output operation (that is, DMA) is allowed out of a shadow page.

Memory accesses for servicing SC misses have a priority lower than that of compulsory misses on the data caches, but a higher priority than instruction misses and prefetching requests. If a branch misprediction is discovered, memory accesses triggered by SC fetches along the mispredicted path are canceled and the appropriate pipeline stages in the CHG are also flushed. Interrupts are handled like branch mispredictions. Interrupts flush instructions and results of the earliest basic block in the pipeline (that has not been validated) and resumes from the beginning of the basic block if the interrupt was generated by the BB itself. External interrupts are handled after completing validation of the current BB. The context structure in the OS holds the base address of the signature table in the RAM. A signature base register is used to point to the starting (=base) address of the

RAM-resident signature table of the executing module (Sec. V).

B. Supporting Cross-Module Calls

A single program module may call functions within a number of other independently compiled modules. These modules include shared components as well as statically or dynamically linked libraries/modules. Each such module will have its own encrypted signature table. As execution switches from one module to another, the register pointing to the base of the signature table has to be switched from that of the calling module's to that of the callee's. The specific hardware support for cross module calls is part of the signature address generation unit (**SAG**) shown in Figure 1 and consists of a set of B base registers that contain the base addresses of the RAM-resident signature tables for up to B called modules. Associated with each such base register is a pair of limit registers that record the starting and the last virtual addresses of the corresponding module and a key register that holds the secret key used for decrypting the corresponding reference signature.

For statically-linked modules, the base registers in the SAG, the associated limit register pair and the key register are filled in by the linker (which is trusted). For dynamically linked modules, the base and limit register pairs and the key register are initialized on the first call to the dynamically linked module, when the jump vectors for the functions within the called dynamically-linked module is initialized. The usual (and trusted) dynamic linking function is extended to handle the base and limit-register pairs and the key register. At every call or return instruction, the address of the target is associatively compared against the addresses stored in all of the B limit register pairs. The base register for the signature table to use is the one whose corresponding limit register values enclose the called function's entry (or return) address.

The cross-module call support meets any remaining part of Requirement R1 that is not met by the SC and its associated logic. The Silicon requirements for B base address and limit register pairs and their associated comparators are fairly modest for small values of B (16 to 32). When more than B modules are involved, an exception is generated when none of the limit register pairs encloses the called/returned address and the exception handler is invoked to manage the base-limit registers.

C. Signature Cache (SC) Details

The SC is a set-associative cache that is accessed using the basic block (BB) address, which is the address of the last instruction terminating the BB. An entry in the SC (Figure 2) contains a field to indicate the entry type, actual branch outcomes that direct control to flow out of the block, the decrypted crypto hash of all of the instructions in the BB and the address of a successor or predecessor BB (for BBs that are the target of a return instruction). Static branches (whose possible targets do not change with execution, such as a PC-relative conditional branch) are thus verified implicitly using the crypto hash of the BB. If a BB has more than one successor or one predecessor, only the entries for the most recently used branches are maintained within the SC entry. This requires the use of additional logic to handle replacements of the successor and predecessor address fields within an SC entry.

As instructions of a BB are fetched along the speculated execution path, they are fed into the CHG pipeline to compute the crypto-hash of the instructions in the BB. To permit the flushing of entries in the CHG, the inputs are tagged with the id of the successor basic block along the predicted path. Prior to this, using the address of the BB (address of the instruction that terminates a BB) as the lookup key, the signature cache is probed to determine if the decrypted reference signature and the address of the successor basic block are available in the SC. If both are available, we have a SC hit and no further actions are needed. Two situations arise on a SC miss: one is a *partial miss* that indicates the presence of a matching entry with a decrypted reference signature but with no address listed for the successor basic block encountered along the predicted path. The other situation is a *complete miss*, where no matching entry is found in the SC. In either of these two SC miss scenarios, a memory access, going through the normal cache hierarchy, is triggered to fetch the missing information into the SC. The authentication check in REV is invoked when the last instruction in a basic block commits. This check uses the generated hash signature, address and outcome of the successor basic block for the authentication match the information stored in the SC entry.

Figure 3 summarizes how the SC is used to validate the execution of a BB that is terminated with a computed branch. For BBs that ends with a non-computed branch, a partial

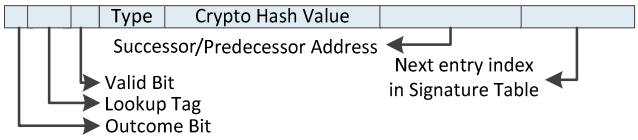

Figure 2. Signature Cache Entry Structure

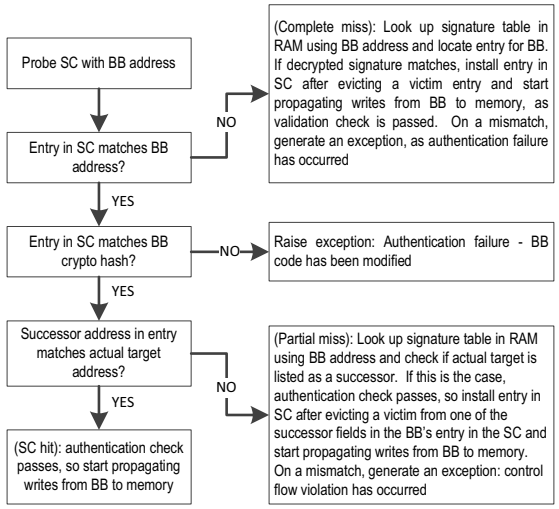

Figure 3. Overview of SC operations in authenticating a BB ending with a computed branch

miss is not possible. BBs ending with a return are validated using the SC as described in Section V.A. The data for the encrypted control flow signatures stored is in the RAM as a hash-indexed table and this table is organized and accessed as described in Section V.

D. Handling Computed Branches

REV relies on the generation of a reference control flow graph (CFG) prior to the execution and thus all the computed branch addresses must be identified. REV treats any unidentified computed branch address as illegal. There are two broad ways to extract the legitimate source-target pairs for the indirect branch addresses [46]: extracting through static analysis [30, 57] or by performing program-profiling runs, as many model-based solutions have done [24, 69]. Alternatively, there are tools to generate full CFG that correctly identifies the source and target addresses for dynamic functions and calls. Abadi et al. shows how Vulcan [51] is used to extract the full CFG that includes the computed branches [1, 2]. An alternative hardware mechanism [39] also uses profiling runs to train a mechanism to just authenticate indirect branches. We used static analysis and profiling runs for some of the SPEC benchmarks to identify the targets of indirect branches.

E. Handling Legitimate Self-Modifying Code

REV relies on basic block signatures derived from a static analysis and as such it cannot directly handle intentional dynamic binary modifications (self-modifying code). It is useful to note that such modifications are a perpetual threat to system security, as noted in several recent and past literatures [10, 54, 29, 66]. However, there are real scenarios where binary modification is used intentionally for reasons of efficiency, such as by OS boot loaders and by Java just-in-time compilers.

In theory, the generated code's signature can be produced for REV's use before its execution but the overhead will be significant. Several options are available here. One option would be to only validate the code (such as JIT) performing the run-time modifications. The actual modified code cannot be validated. When such modified binaries are running, the REV mechanism can be momentarily disabled by the OS. Execution containment or sandboxing [5] can be used to limit potential violations from the generated code. As another option, one can also use a provably secure code modifier [63] or use alternative dynamic validation techniques [26] to ensure that the generated code is safe. REV can thus rely on these alternatives for handling dynamically generated code. Similarly, the signature of dynamically linked drivers (based on dynamically-bound address vectors) can be generated by the OS prior to their deployment and use.

REV will certainly be useful in authenticating the execution of OS, libraries and utilities as well as applications that do not rely heavily or at all on run-time code modifications - thus it provides platform level guarantees that are not available today.

V. Reference Signature Table in RAM

Each executable module has its own memory-resident ***signature table***. The contents of this table are encrypted (Section IV.A) and each entry corresponds to a basic block (BB). Each entry in this table holds the 4 bytes of crypto hash signature of the BB and predecessor and successor information. The format used exploits the fact that most BBs (other than those ending with computed branches and the return instructions) have only a few targets. Since we verify the integrity of the committed instruction in the BB, there is no need to verify the target addresses for the non-computed branches that terminate a BB (Sec. IV.A). In contrast, the target addresses of the computed branches returns can be altered even if the instructions in the BB are genuine, so the target addresses for the computed branches and returns need to be verified explicitly.

The signature table entry for a conditional branch will include its two possible targets, while that for an unconditional jump or a call instruction will have a single target. However, for a computed branch, all potential target addresses need to be listed in the entry. Likewise for a function that is called from multiple locations, the signature table entry for the return instruction terminating such a function should list multiple targets.

REV uses a hashed table organization for the signature table in the RAM. Recall that the address of the last instruction in the basic block is used to identify a basic block. This address A is used to locate a matching entry in the signature table, deriving an index A mod P, where P is chosen to minimize the conflict in the hash table.

A. Delayed validation of returns

For an efficient implementation the signature table lookup on a SC miss, the entries located using the hash indices need to be uniform in size. If we restrict each entry located using a hash index to have at most one target addresses, then a linked list needs to be used to hold the remaining target or predecessor addresses for of control flow instruction that has more than one target or one predecessor address. Traversing such a list to locate a matching target address requires indirection through memory locations and this will prolong the SC miss handling time. To avoid this performance penalty, for return instructions that terminate a popularly called function, a BB ending with a return is validated in two steps: (a) we only validate the crypto hash of the signature of the BB terminated by the return instruction but save the address of the return instruction in a special latch internal to the SC mechanism; (b) when the control flow instruction that terminates the first basic block (say, RB) entered in the calling function following a return instruction, say R, is validated, we validate the execution of the RB in the usual way and simultaneously the address of R is validated. The expected address of R is stored as part of the entry for RB. Recursive functions are handled in a slightly different fashion.

B. Format of signature table entries

Each entry in the signature table holds one target BB address, one predecessor BB address (if any) and the crypto

hash value for the basic block. Since there is only one target BB and one predecessor BB address in each entry, to reduce the size of each entry, the BB crypto hash includes these addresses along with the address of the BB and instructions in the BB. Three tag fields, consisting of the lower order bits of the addresses of the BB, its target and its predecessor are used to uniquely identify the correct table entry and discriminate it from entries for other BB entries that share the same hash table index.

The generic structure on an entry in the memory resident table (decrypted) is shown in Figure 4. An entry has a type tag field indicating the entry type, the crypto hash of the BB, the address or equivalent information (such as a branch offset) to determine the address of the first instruction in the successor BBs. As an entry in the memory resident signature table is located using a hash indexing scheme, there is a chain of entries sharing a common index that need to be walked for locating the matching crypto hash value. Since the BB crypto hash value includes the crypto hash of the instructions in the BB, the address of the BB, the target branch address and the predecessor BB address, it is very unlikely that someone can forge the crypto hash value for the entry at specific index. In our experiments, the crypto hash used was found to be unique. However, it is impossible to prove that such hashes are always guaranteed to be unique. The three tag fields in the entry are used to further distinguish among BBs that may have the same crypto hash value, an extremely unlikely situation.

The "Next Entry" field in the entry contains a pointer to a spill area that lists additional successors and/or the address of a preceding return instruction (the last applicable only to BBs entered on a return) and the next entry, if any, that shares the same hash index (link to the so-called collision chain). The last element in an array of spill values associated with a single table entry is identified using a special tag value.

The primary entry located using a hash index and the following collision chain links are identical in size, and depending on the entry type, some of the fields are unused. Wherever possible, these unused areas are used to serve as a spill space. Other techniques are also used to minimize the space requirement of the entries, such as using offsets instead of full addresses, listing branch outcomes implicitly, etc. The signature table sizes for the SPEC executables we used ranged from about 15% to 52% of the executable size with the average of 37%. Thus, we are trading off memory space for the benefit of complete validation of program execution at run-time. The signature address generation unit (Figure 1) generates a memory address for the required signature table entry on a SC miss, based on the signature table's base address in the RAM (obtained from the signature address base register of the executing module, Sec. IV.A) and the hash of the address of the control flow instruction that terminates a BB (obtained from the front-end stages). Once the entry is retrieved, based on the type of the signature, the crypto-hash of the committed instructions, target address and the predecessor addresses listed in the entry are compared with the actual generated values at run-time. On a mismatch, additional entries in the spill area are progressively looked up following the link listed in the entry. If no matching entry is located, control is presumably flowing on an illegal path and an exception is generated.

C. *Aggressive CFA of every branch*

The crypto hash value used to identify each basic block (BB) must be able to detect any change that is made to the BB. Usually a crypto hash value will vary between 16 to 32 bytes. To cut down on the storage requirement, we used the last 4 bytes of the crypto hash value to identify basic blocks uniquely. While we did not see any issue with this in our experiments, in extreme cases and for long BBs it may be possible to alter the BB in a way so that the last 4 Bytes in the crypto hash for the modified BB hash value to be identical to the last 4 Bytes of the crypto hash for the original BB. For validating control flow in such extreme cases, control flow must be validated not just on the computed branches but for every branch.

The format of an entry in the memory resident signature table (in decrypted form) for dealing with this extreme situation is shown in Figure 5. Here, since we verify every branch target addresses, we include both possible target addresses in the signature. There will be multiple entries connected to each other via a linked list for those branches that have more than two target addresses. Unfortunately, the price that has to be paid for this full coverage is in the form of an increase in the table size, ranging from 40% to 65% of the executable size, which is almost double of the signature table sizes that we have in our original design.

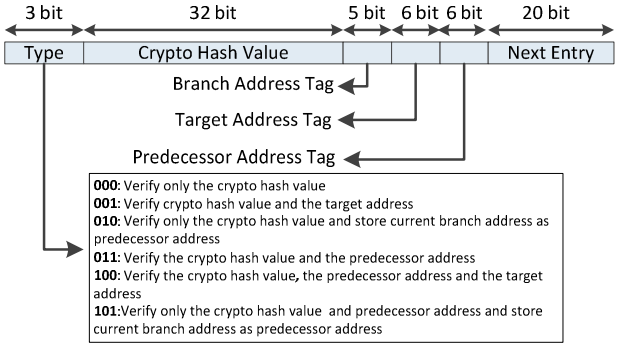

Figure 4. Signature Table Entry Structure

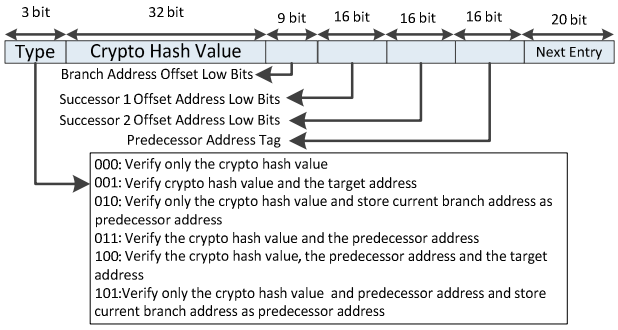

Figure 5. Signature Table Entry for Aggressive CFA

D. Validating control flow integrity only

If we assume that the system is protected against code integrity attacks, we can use a smaller signature table. REV can be easily adopted to only validate control flow integrity, removing the need to store, generate and use the crypto hashes of the BB for validation. Here, each entry in the table includes the full address of the target, 12 bits for the source address as a discriminator tag and 20 bits for the next index address (in case there are more than one successor for the BB corresponding to the entry). This leads to signature table sizes that range from 3% to 20% of the binary size across the SPEC benchmarks, with the average signature table size that is only 9% of the executable size. A static analysis of the SPEC benchmarks shows the dynamic branches, including the returns, constitute only about 10% of total number of all branch instructions on the average.

VI. Timing and Other Implications

The REV crypto generator (CHG, in Figure 1) starts generating the hash of a basic block as soon as instructions are fetched along the predicted path. This crypto hash value is needed for validating the instructions in the basic block after the last instruction in the basic block is committed. To completely overlap the delay of crypto hash generation, say H, with normal pipeline operations, one must ensure that in the worst case that H = S, where S is the number of pipeline stages in-between the final instruction fetch stage and the commit stage. In a typical high-end X86 implementation, S ranges from 12 to 22. In reality, instruction commitment can be delayed by additional cycles in the reorder buffer. For our simulations, we assumed that S is 16 and assumed the worst-case value of H as 16. Some SHA-3 candidates, including the cube hash algorithm can meet the latency requirement described above. For instance, a cube hash implementation with 5 rounds can meet the 16 cycle latency goal with parallel pipelines [17, 59]. For a 5-round cube hash algorithm, we also found that the crypto hash generated for the basic blocks of the SPEC 2006 benchmarks were unique. Alternatives to the cube hash are also presented in [50, 23, 59], that describe pipelined hash generators with a latency ranging from 14 to 21 cycles, particularly the designs presented in [59] for 180 nm implementations. These designs can be further refined for lower latencies in contemporary 32 nm technology and realize a CHG that will not hold up usual commitments on a SC hit, an assumption that we have made in our simulations.

If the latency H of the CHG is higher, we can add dummy post-commit stages to the pipeline to effectively increase S to equal H. On a SC miss, whether it's partial or full (Section IV.C), pipeline stalls occur and such stalls are actually modeled in full detail in our simulator.

The areas of the 32 Kbyte SC, registers, latches, comparators, RAM array implementing the write queue extension were estimated using CACTI 6.0 [40]. The area and energy dissipation of the CHG was extrapolated from 180 nm to 32 nm technology from the area and power data given in [59]. For a 3 GHz. pipeline clock frequency, the dynamic power overhead of REV is estimated to be about 7.2% of the power consumption of the core in the base design (with a private L1 and L2 cache). The base case core power is estimated from McPAT [37]. The area overhead added by REV is about 8% of the base core's area. With a shared L3 cache and the I/O pad power added in, the overall power overhead added by the REV logic to a multicore chip is reduced from 7.2% to less than 5.5%. In our opinion, the area and power overhead for REV, retrofitted into a contemporary out-of-order processor, is acceptable. Note also that encryption/decryption units (AES and others) are already on chip on newer CPUs, including embedded CPUs for cell phones. If the decryption logic for REV is shared with the main CPU, then the estimated die area and power consumption overhead will be even lower than what we have estimated above, both at the level of the core and at the level of an entire multicore chip.

VII. Limitations and Security of REV

REV relies on the use of encrypted reference signatures for basic blocks to be stored in the RAM. This requires the use of secret keys for decrypting the reference signatures for each executable module. Such keys can be held in secure key storage in the RAM that can be implemented using a TPM-like system implemented inside the CPU. The external TPM mechanism in contemporary system is not usable, as it remains vulnerable to bus snooping and furthermore the bus speeds inhibit its continuous use at rates that match the (unaltered) CPU's throughput [60]. REV thus assumes that the CPU has an internal TPM-like mechanism to be used for attestation and key management. Each signature table can also be encrypted with a different symmetric key (Section IX). An adversary may overwrite the memory locations for key storage - at best this will cause the validation to fail (Section IV.A) but this will never allow illicit code to be authenticated, as the secret key for decrypting the reference signature is stored either encrypted in the memory or decrypted inside the CPU.

In REV, we assume that the linker/loader of the application is trusted hence the signature tables are loaded correctly and the registers within the SAG are initialized at the beginning of the execution.

REV also requires two system calls. One of these is for loading the base addresses and decryption key of the memory areas that contain the encrypted reference signatures into the special registers within the crypto hash address generation unit. The second system call is used to enable or disable the REV mechanism and this is only used when safe, self-modifying executables are running (Section IV.E). For REV to work properly, these two system calls must be secured as well.

Like any technique that detects control flow compromises of a general nature at run-time, REV requires a priori knowledge of the targets of computed jumps and this is a potential limitation to its use as a solves-all panacea. However, recent advances in static code analysis [51, 2] as well as the use of profiling runs can be used to identify the targets before execution to overcome this limitation. In the least, REV can be used to protect critical software in a typical system, such as the OS, libraries, utilities etc., where

the targets of computed jumps can be determined exhaustively.

We believe our approach introduces a relatively simple hardware support for preventing any code injection attacks – either direct or indirect as well as any attack that cause the control flow to be altered. We also believe that our TCB (trusted computing base) is small hence it is less likely to have a vulnerability. Table 1 shows some well-known attacks and how REV handles them.

VIII. Experimental Assessments

The performance overhead of the dynamic validation of executions with REV was evaluated through simulation using the cycle-accurate MARSS full system microarchitectural simulator for X86-64 ISA [45]. MARSS models an out-of-order superscalar processor with two levels of private caches. This simulator was modified extensively to correctly model cache contents and a reasonably realistic memory system and system bus, including DMA bursts, memory banks and faster accesses to open DRAM pages. Table 2 summarizes the pertinent configuration details. For each of the simulated benchmarks, we measured user-level IPC (instructions committed per clock cycle) in the course of committing 2 billion instructions of the benchmarks. For each benchmark we used the harmonic mean of results from 5 runs. TLBs were also modeled correctly, including the TLB usage in the course of servicing SC misses. We used statically linked executables for our studies. The SC accesses the memory via the same cache hierarchy that the CPU uses. An additional L1-D cache port was assumed for use by the SC.

Prior to presenting the results for the REV scheme, we give some pertinent basic block statistics need to be presented. Across the SPEC CPU 2006 benchmarks, the number of basic blocks range from 20266 for mcf to 92218 for gamess. The average number of successors per basic block range from 1.68 for soplex to 3.339 for gamess. The average number of instructions per basic block range from 5.5 (for mcf) to 10.02 (for gamess).

Figure 6 shows the IPCs realized on the SPEC 2006 benchmarks for the base case (with no support for authenticating executions) and on a base case processor augmented with REV with 4-way set-associative signature caches of capacity 32 Kbytes and 64 Kbytes, respectively. Figure 7 depicts the performance losses compared to the base case design (as a percentage of the base case IPC) for the REV technique. As seen from Figure 7, for many benchmarks (such as bzip2, cactusADM, dealII etc.) the performance penalty with REV is negligible, while for some other benchmarks (like gcc, gobmk, h264ref) the IPC overhead of REV is relatively high. The performance penalty with the incorporation of REV into the baseline OOO design, compared to the baseline, is 1.87% on the average across all benchmarks. Expectedly, the IPC overhead drops as the size of the SC increases, as the number of misses in accessing the signature cache during authentication checks drop with an increase in the SC size, reducing commit stalls. For explanations of the results shown in Figure 7, it is instructive to examine the number of

TABLE II. Processor and Memory System Configuration

Fetch Queue Size	32	Unified Register File	256 registers
LSQ size	92	L1D size, latency, associativity	64 Kbytes, 2 cycles, 4
Dispatch Width	4	L1I size, latency, associativity	64 Kbytes, 2 cycles, 4
ROB size	128	L2 size, latency, associativity	512 Kbytes, 5 cycles, 8
Function Units	2 ALU , 2 FPU, 2 store + 2 load units	Memory latency, Banks, DMA channels, burst size	100 cycles for first chunk, 8, 64 ch, 64-Byte bursts
TLBs, branch predictor	32 entry L1 I-TLB and 128 entry L1 D TLB, each backed by a 512 entry L2 TLBs; 32K Gshare. DTLB shared with SC using an extra port.		

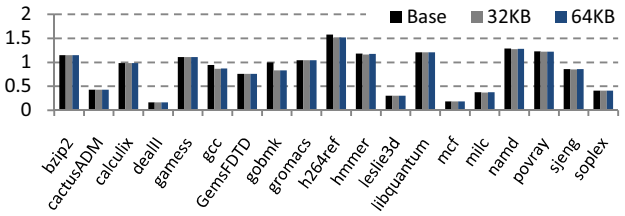

Figure 6. IPCs for the base case and REV for SCs of 32 Kbyte and 64 Kbyte

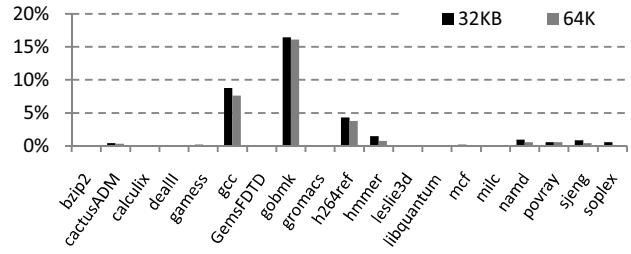

Figure 7. IPC(Instructions per Cycle) overhead in % of the benchmarks for different SC sizes for REV

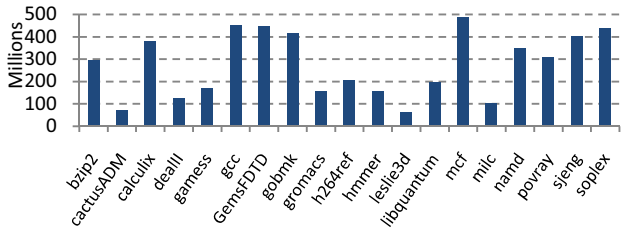

Figure 8. Number of committed branches during execution

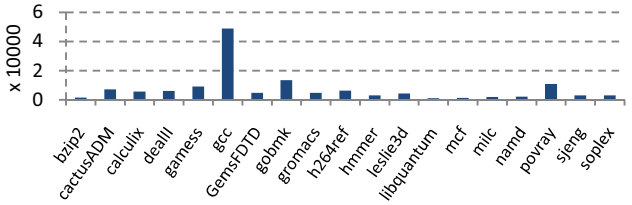

Figure 9. Number of unique branches during execution

committed branches (Figure 8), the number of unique branches encountered during execution (Figure 9) and the cache access statistics, as access counts (Figure 10 and Figure 11) in the course of SC misses (which are serviced using the existing memory hierarchy), all for a 32 KByte SC. For gcc, both the number of unique branches encountered and the total number of committed branches are very high compared to the other benchmarks, and to make things worse, the cache miss rates in servicing a SC miss are also high.

All of these factors result in high SC miss counts and significant pipeline stalls occur when instructions cannot be committed until the SC miss is handled. A similar situation is seen for gobmk. Further, gobmk has more SC misses and more L1 misses than gcc. As a result, gobmk has the highest performance overhead with REV.

In general, benchmarks with reduced control flow locality will have higher SC misses and a higher performance overhead. Benchmarks gobmk, gcc, h264ref, hmmer have some overhead that are correlated to their SC miss counts. All benchmarks other than gcc and gobmk still have less than 5% performance overhead.

For some benchmarks like mcf, the higher branch counts that can potentially increase the overhead are compensated by higher hit counts on SC and as a result, the overall performance overhead is still negligible. The overall performance overhead remains acceptable, less than 1%. Benchmarks like bzip2, cactusADM, calculix, deal, hmmer, leslie3d, libquantum, mcf, milc soplex, sjeng all have a small set of unique branch addresses and very low SC miss rate and this reduces their overall IPC overhead. While soplex has higher number of committed branches than the sjeng, it has lower L1 miss rate thus soplex has a lower overhead than sjeng. The main takeaway from the results for REV is that a small 32 Kbyte SC enables the average IPC overhead to be kept at acceptable levels.

In summary, the performance overhead of REV in the simulated execution of the SPEC benchmarks was limited to single digit percentages in all cases except for gobmk. Only for gobmk, the overhead was about 15%. The average performance overhead of REV across all SPEC benchmarks was 1.87% for a 32 Kbyte signature cache and 1.63% for a 64 Kbyte signature cache.

Figure 12 shows the performance overhead for REV when we perform aggressive validation using the larger signature tables. In this case, we get a slightly better performance because now we can verify the addresses of up to two successors using a single entry.

At run-time, only about 1% to 10% of the all executed branches are computed branches, resulting in considerably fewer accesses to the signature table and culminating in only a 0.04 % to 1.68% performance overhead across the SPEC benchmarks for CFI-only validations.

IX. Key Management

REV assumes a TPM-like attestation inside the CPU. This attestation helps us in exchanging keys securely without revealing the key to the memory at any point. Signature tables are encrypted using symmetric keys and the

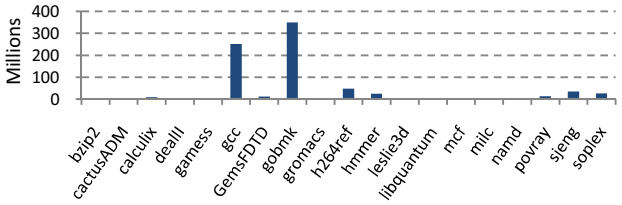

Figure 10. Signature cache miss counts

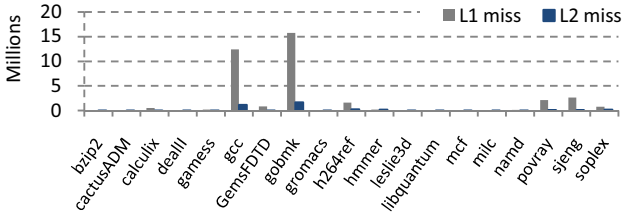

Figure 11. Cache miss statistics when servicing SC misses

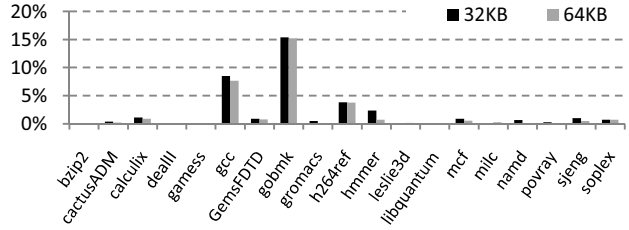

Figure 12. IPC overhead in % for the benchmarks for different SC sizes for REV with aggressive validation

symmetric key is encrypted with a public key specific to the CPU. The encrypted symmetric key is stored at the beginning of the signature table. This key is encrypted on a per CPU basis. The signature tables can be re-encrypted with different symmetric keys by a trusted entity. Since the signature keys can be decrypted only by the CPU associated with the public key, the actual keys for decrypting the signature tables always remain safely within the CPU and are not exposed in memory.

X. Conclusions

REV appears to be a viable solution for detecting control flow and code integrity attacks at run-time in a modern OOO processor with a low overhead. Binary modifications or ISA extensions are not needed. REV authenticates execution at the granularity of basic blocks using statically derived reference signatures that are stored in an encrypted form; memory updates from a basic block take place only after its execution is authenticated. Self-modifying code is handled in REV in two possible ways (Sec. IV.E). REV requires target addresses of control flow instructions to be determined statically. When this is not possible, alternatives are suggested (Sec. IV.D and IV.E). In general, REV can be used to protect critical infrastructure code like the OS, libraries, utilities, network stack etc, where code

modification, if any, is limited and trusted and where targets of indirect jumps can be determined through exhaustive processes. REV thus represents a viable way of building trusted systems that are dynamically and continuously validated. Alternative uses of the REV mechanism are possible but not discussed here. For example, failed validation attempts can reveal signatures of the offending code that can be used to detect them later.

ACKNOWLEDGEMENTS

This work is supported in part by NSF Award Nos. 1040666, 0958501 and by an award from Intel Corporation. No endorsement of this work by the sponsors is implied in any form.

REFERENCES

[1] Abadi, M., Budiu, M., Erlingsson, U. and Ligatti, J., "Control-Flow Integrity: Principles, Implementations, and Applications", Proc. ACM Conference on Computer and Communications Security (CCS), pp340-343, 2005.

[2] Abadi M., Budiu M., Erlingsson U., Ligatti J., "Control-flow integrity principles, implementations, and applications" ,ACM Transactions on Information and System Security (TISSEC), Vol. 13 Issue 1,pp. 1-40, 2009.

[3] Aktas, E., and Ghose, K., "DARE: A framework for dynamic authentication of remote executions", in Proc. of Annual Computer Security Applications Conference (ACSAC), pp. 453-462, 2008.

[4] Aktas, E., and Ghose, K., "Run-time Control Flow Authentication: An Assessment on Contemporary X86 Platforms", ACM Applied Computing Conference, Security Track (SEC@SAC), pp. 1859-1866, 2013.

[5] Ansel, J., Marchenko, P., Erlingsson, Ú., Taylor, E., Chen, B., Schuff, D. L.,Yee, B, "Language-independent sandboxing of just-in-time compilation and self-modifying code", in Proc. of 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation,pp. 355-366, 2011.

[6] Arora D.,Rav S., Raghunathan A., Jha N., "Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors", IEEE TVLSI, pp. 1295-1308, 2006 and earlier version in Proc. DATE.

[7] ASLR description on web pages at: http://pax.grsecurity.net/docs/aslr.txt.

[8] Bletsch T., Jiang, X., Freeh V., "Mitigating code-reuse attacks with control-flow locking", Proc. of the 27th Annual Computer Security Applications Conf. (ACSAC), pp. 353-362, 2011.

[9] Bletsch T., Jiang, X., Freeh V. W., Liang Z., "Jump oriented programming: a new class of code-reuse attack", In Proc. of the 6th ACM Symposium on Information, Computer and Communications Security. ASIACCS, pp. 30-40, 2011.

[10] Blazakis D., Interpreter Exploitation: Pointer Inference and JIT spraying", available on the web "at http://www.semantiscope.com/research/BHDC2010/BHDC-2010Paper.pdf.

[11] Baldi M., Ofek Y., Yung M., "The TrustedFlow Protocol: Idiosyncratic Signatures for Authenticated Execution", in Proc. IEEE Workshop on Information Assurance, pp. 288-289, 2003.

[12] Baldi M., Ofek Y., Yung M., "Idiosyncratic Signatures for Authenticated Execution, TheTrustedFlow, Protocol and its Application to TCP", in Proc. Sym. on Comm. Systems and Networks (CSN), pp. 204-206, 2003.

[13] Brad S., "On exploiting null ptr derefs, disabling SELinux, and silently fixed Linux vulnerabilities", Dailydave List; 2008. http://grsecurity.net/~spender/exploit.tgz.

[14] Buchanan E., Roemer R.,Shacham H., Savage S., "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC", in In Proc. of the 15th ACM conference on Computer and communications security, ACM CCS, pp. 27-38, 2008.

[15] Checkoway S., Davi L., Dmitrienko A., Sadeghi A., Shacham H., Winandy M., "Return-oriented programming without returns", In Proc. of the 17th ACM conference on Computer and communications security, ACM CCS, pp. 559-572, 2010.

[16] Chen, B. and Morris, R., "Certifying Program Execution with Secure Processors", HotOS Conf., pp. 133-138, 2003.

[17] Cube Hash web pages at: http://cubehash.cr.yp.to/, see also Bernstein's presentation available at: http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/presentations/BERNSTEIN_3-cubehash.pdf.

[18] Chen P., Xing X., Han H., Mao B., Xie L., "Drop: Detecting return-oriented programming malicious code", In Proc. of ICISS, pages 163–177, 2009.

[19] Chen P., Xing X., Han H., Mao B., Xie L., "Efficient detection of the return-oriented programming malicious code", ICISS'10 Proc. of the 6th Int'l. Conf. on Information Systems Security, pp. 140-155 , 2010

[20] Fiskiran A., Lee R., "Runtime Execution Monitoring (REM) to Detect and Prevent Malicious Code Execution.", in Proc. of the IEEE Int'l Conf. on Computer Design, pp. 452-457, 2004.

[21] Flamer: Highly Sophisticated and Discreet Threat Targets the Middle East http://www.symantec.com/connect/blogs/flamer-highly-sophisticated-and-discreet-threat-targets-middle-east.

[22] Floating Licence Management, A Review of Flexlm available at: http://wob.iai.uni bonn.de/Wob/images/36311141.pdf, 2006.

[23] Guo X, Huang S., Nazhandali L.,Schaumont P., "Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC Implementations", in Proc. the 2nd. SHA-3 Candidate Conference, organized by NIST, 2010, at: http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/SCHAUMONT_SHA3.pdf.

[24] Giffin J., Jha S.,Miller B. , "Efficient context-sensitive intrusion detection", in Proc. 11th NDSS, 2004.

[25] Gelbart, I. , Ott , P., Narahari , B. Simba R. Choudhary A., Zambreno J., "Codesseal: Compiler/FPGA approach to secure applications", in Proc. of IEEE International Conf. on Intelligence and Security Informatics, pp. 530-535 , 2005.

[26] Homescu A., Brunthaler S., Larsen P., and Franz M., "Librando: transparent code randomization for just-in-time compilers", in Proc. of the 2013 ACM SIGSAC conference on Computer & communications security (CCS), pp. 993-1004, 2013.

[27] Huang R., Deng D., Suh E., "Orthrus: efficient software integrity protection on multi-cores", in Proc. of architectural support for programming languages and operating systems, ASPLOS, Vol. 38, No. 1, pp. 371-384 , 2010.

[28] Hund R., Holz T. ,Freiling F, "Return oriented rootkits: Bypassing kernel code integrity protection mechanisms", in Proc. of Usenix Security Symposium, pp. 383-398, 2009.

[29] Hu W., Hiser J., Williams D, Filipi A., Davidson J., Evans D., Knight J., Nguyen-Tuong A., Rowanhill J., , "Secure and Practical Defense Against Code-injection Attacks using Software Dynamic Translation" , In Proc. of the 2nd international conference on Virtual execution environments (pp. 2-12). , 2006.

[30] Hind M., Pioli A.. "Which pointer analysis should I use?", in Proc. of the International Symposium on Software Testing and Analysis, Vol. 25, No. 5, pp. 113-123, 2000.

[31] IBM Corporation, IBM 480X series PCIe Cryptographic Coprocessor product overview, available at: http://www03.ibm.com/security/cryptocards/pciecc/overview.shtml.

[32] Ittai A., Shay G., Simon P Johnson, Vincent R S., "Innovative Technology for CPU Based Attestation and Sealing", http://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing.

[33] Kauer, B., "OSLO: Improving the Security of Trusted Computing", in Proc. Proc. of 16th USENIX Security Symp, p.1-9, 2007.

[34] Kennell R., and Jamieson, L. H., “Establishing the genuinity of remote computer systems”, in Proc. 12th USENIX Security Symp, pp. 295-310, 2003.

[35] Kayaalp M., Ozsoy M., Abu-Ghazaleh N., Ponomarev D., "Branch Regulation: Low Overhead Protection Fron Code Reuse Attacks", 39th Int'l. Symp. on Computer Architecture (ISCA), pp. 94-105, 2012.

[36] Kayaalp M., Schmitt T., Nomani J., Ponomarev D and Abu-Ghazaleh N, "SCRAP: Architecture for Signature-Based Protection from Code Reuse Attacks", The 19th IEEE Int’l Symp. on High Performance Computer Architecture (HPCA), pp. 258-269, 2013.

[37] Li S., Ahn J., Strong D., Brockman J., Tullsen D, and Jouppi N., “McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures”. In Proc. of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42), pp. 469-480, 2009.

[38] Lee R., Kwan P., McGregor J., Dwoskin J., Wang Z., “Architecture for Protecting Critical Secrets in Microprocessors”, in Proc. of the Int'l. Symp. on Computer Architecture (ISCA), pp. 2-13, 2005.

[39] Lee G., Shi Y., Lin H., “Indirect Branch Validation Unit”, in Microprocessors and Microsystems, Vol. 33, No. 7-8, pp. 461-468, 2009.

[40] Muralimanohar, N., Balasubramonian, R., & Jouppi, N. P. CACTI 6.0: A tool to model large caches. HP Laboratories. 2009.

[41] Meixner A.,Bauer M. E.,Sorin D., “Argus: Low Cost, Comprehensive Error Detection in Simple Cores”, In Proc. of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42), pp. 210-222, 2007.

[42] Nagarajan V., Gupta R., “Architectural Support for Shadow Memory in Multiprocessors”, in Proc. ACM int’l Conf. on Virtual Execution environments (VEE) , pp. 1-10, 2009.

[43] Owusu E., Guajardo J., McCune J., Newsome J. , Perrig A., Vasudevan A, , “OASIS: on achieving a sanctuary for integrity and secrecy on untrusted platforms”,Proc. of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 13-24, 2013.

[44] Pappas V, “kBouncer: Efficient and Transparent ROP Mitigation”, Microsoft Blue Hat Competition Winner, 2013.

[45] Patel, A., Afram, F., Chen, S., Ghose, K. MARSS: a full system simulator for multicore x86 CPUs. In Proc. of the 48th Design Automation Conference pp. 1050-1055, 2011.

[46] Park Y., “Efficient Validation of Control Flow Integrity for Enhancing Computer System Security”, PhD dissertation, Iowa State Univ., 2010.

[47] PaX webpages at: http://pax.grsecurity.net/

[48] Pappas V., Polychronakis M., Keromytis A., “Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization” Proc. IEE Security and Privavcy Symposium, pp. 601-615 , 2012.

[49] Ragel R., Parameswaran S., “Impres: integrated monitoring for processor reliability and security”, In 43rd ACM/IEEE Design Automation Conference, pp. 502-505, 2006.

[50] Satoh, A., “ASIC Hardware Implementations for 512 bit Hash Function Whirlpool”, in Proc. Int. Conf. on Circuits and Systems, pp. 2917-2920, 2008.

[51] Srivastava A., Edwards A., and Vo. H., “Vulcan: Binary Transformation in a Distributed Environment.”, Tech. Report MSR-TR-2001-50, Microsoft Research, 2001.

[52] Shacham H. “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)”, In14th The ACM Conference on Computer and Communications Security (CCS), pp. 552-561 , 2007.

[53] Schuette, M. and Shen, J., “Processor Control Flow Monitoring Using Signatured Instruction Streams”, in, IEEE Transactions on Computers, Vol. C-36 No: 3, pp. 264-276, 1987.

[54] Salamat B., Jackson T., Wagner G., Wimmer C., Franz M., “Runtime Defense against Code Injection Attacks Using Replicated Execution”, in IEEE Trans. on Dependable and Secure Computing, Vol. 8 No:4, 588-601, 2011.

[55] Seshadri A., Luk M., Shi E. Perrig A., Doorn L., Khosla P., “Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems”, in ACM Symp. on Operating Systems Principles,pp. 1-16, 2005.

[56] Suh, E.G., O'Donnell. C.W., and Devadas, S., "Aegis: A Single-Chip Secure Processor", IEEE Design and Test of Computers, 24(6), pp. 63-73, 2007.

[57] Steensgaard B., “Points-to analysis in almost linear time”, In Proc. of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages pp. 32-41, 1996.

[58] Stuxnet Dossier http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[59] Tillich, S., Feldhofer, M., Kirschbaum, M., Plos T, Schmidt J., Szekely A.,., “Uniform Evaluation of Hardware Implementations of the Round Two SHA-3 Candidates”, in Proc. the 2nd. SHA-3 Candidate Conference, organized by NIST, 2010.

[60] Trusted Platform Module spec. at: http://www.trustedcomputinggroup.org/.

[61] Wang Z., Jiang X., “HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Inte, grity”, in Proc. of the IEEE Symposium on Security and Privacy, pp. 380-395, 2010.

[62] Wartell R., Mohan V., Hamlen K., Lin Z.,“Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code”, Proc. ACM Conf. on Computer and Communications Security (CCS), pp. 157-168, 2012.

[63] Wei T., Wang T., Duan L., and Luo J., “Secure dynamic code generation against spraying”, in Proc. of the 17th ACM conference on Computer and communications security (CCS), pp. 738-740, 2010.

[64] X86-64 buffer overflow exploits described at: http://www.suse.de/~krahmer/no-nx.pdf.

[65] Xia Y., Liu Y.,Chen H., Zang B., "CFIMon: Detecting Violation of Control Flow Integrity using Performance Counters" , 42nd IEEE/IFIP Int’l Conf. on Dependable Systems and Networks (DSN), pp. 1-12, 2012.

[66] Younan Y. , Joosen W. , Piessens F, “Code Injection in C and C++ : A Survey of Vulnerabilities and Countermeasures”, Technical Report, Katholieke Universiteit Leuven Dept. of Comp. Sci, 2004.

[67] Zhang M., Sekar R., “Control flow integrity for COTS binaries”, Proc. of the 22nd USENIX conference on Security, pp. 37-352 , 2013

[68] Zhuang, X., Zhang, T., and, Pande, S., “Hardware Assisted Control Flow Obfuscation for Embedded Processors”, in Proc. Intl. Conf. on Compilers, Architecture and Synthesis for Embedded Systems, pp. 292-302, 2004.

[69] Zhang T., Zhuang X., Pande S., Lee W. , “Anomalous path detection with hardware support”, in Proc. of Int’l Conf. on Compilers, Architectures and Synthesis for Embedded Processors, pp. 43-54, 2005.